REAL-TIME AERIAL VEHICLE DETECTION AND TRACKING WITH DEPTH-AIDED VISION SENSING

BY

BICHENG ZHANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Geir E. Dullerud

ABSTRACT

We study the problem of detecting and tracking flying objects in real-time with color and depth images. We improve the sparse part-based representation learning approach by utilizing depth data from depth vision sensor to achieve much faster detection speed while maintain high detection accuracy. We revised some of algorithms presented in part-based representation method to get marginally better performance. Then we invented a novel data preprocessing method, which is based on edge detection and contour selection to generate possible vehicle locations before the image is processed by classifier. This approach can be applied to any object with distinguishable parts in relatively fixed spatial configurations, and our target here is the flying vehicle at indoor environment. Since flying objects tend to change poses and locations fast and frequently, the detection algorithm needs to run fast so that the tracking algorithm can keep on tracking the detected object. We also use hardware acceleration tools to further increase algorithm speed. The results of vehicle localization and tracking are shown and a critical evaluation of our approaches is also presented.

To my family

ACKNOWLEDGMENTS

When I first time took Prof.Geir Dullerud's lecture, I remembered his flow of obscure mathematical notations and calling the formulas he derived his "baby". I enjoyed the class very much and that is why after a year, I wanted join Geir's research group. Yue Sun is the one introduced me to the lab. We spent day and night crashing first gen HotDrone, and I remember the day we saw HotDrone first time fly steadily, we felt so proud but confused at why it worked at all. Rich's original work turned to be the foundation of my thesis. I literally knew nothing about computer and I kept asking for his help. I knew he was on the edge of running out of patience, but I picked up fast before he got annoyed at me asking, "why it doesn't compile?" Steve handled mechanical and software duty for the drone until I took the job from him and made him jobless. Special thanks for Steve greeting me on Christmas at his home with delicious chili soup. Dr. Qing and Dr. Seungho both getting their PhD degree and their baby congratulations! Thank you two for helping me tackle the difficult problems. Thanks to Wenjia, Weijia, Peter, Di, Yu and everyone work along with me with same passion for the science and technologies. Thanks to my friends, my roommates, and especially my parents for the care and support. Prof. Geir Dullerud, you are my mentor, and will always be. Thank you for everything.

TABLE OF CONTENTS

Chapter 1: Introduction

The object localization and detection is important and is widely used in many areas. The difficulty of developing such algorithm lies in the fact objects tend to appear in complex textured backgrounds with different poses, and the computation overhead for image processing and classification tends to be large to get real-time position information feedback. Thanks to the fast growth of processing power and image sensing technology, the advance of depth sensor and graphics computing unit reduces the cost and computing time, thus open the door for new ways to solve difficult problems.

Our approach to tackle these problems uses the combination of three advanced techniques: the sparse-space part based image representation, depth-image pre-processing and KLT object tracking. The spars-space part based image representation gives us more than 93% detection accuracy rate. The depth-image pre-processing boost detection speed by over 1000 times, which successfully decreases detection time to under 0.3 seconds. KLT object tracking reduces the detection computation overhead and further increase tracking performance. We also tailored some time-consuming algorithm especially for aerial vehicles at indoor environment for marginally better performance. We will describe how we utilize these techniques to achieve real-time detection and tracking on moderate computing hardware.

1.1 Aerial Vehicle Detection

The HotDec Testbed at UIUC is the first state-of-art Networked Testbed (Stubbs & Dullerud, 2001) for hovercraft known as HotDec. It has hosted many experiments including the latest Harbor Attack. With current tremendous interest in the field of Multi-UAV, it should be able to hosting large-scale multiple UAV experiments as well. Because the object tracking and recognition for aerial vehicles has not been developed at HotDec testbed, it is clear that incorporating an advanced surveillance and detection technique is essential for successfully conducting broad range of large-scale aerial vehicles experiments efficiently.

Many different approaches to object detection that use some form of learning have been proposed in the past. In most approaches, images are represented using some set of features, and a learning method is then used to identify regions in the feature space that correspond to the object class of interest. There has been considerable variety in both the types of features used and the learning methods applied.

The precise position tracking of flying vehicle currently requires expensive surrounding vision system with attachment of specific feature points onto the vehicle. Without these equipment setup, it is difficult to track precise location of flying objects. Now with advance of machine learning techniques and modern computation power, we want to achieve real-time flying object position tracking with inexpensive hardware and without help of attachment of feature points.

Thus our major task is to recognize and detect flying objects in complex textured background and keep tracking its location. We will detect the object in images via a sparse, part-based representation (Agarwal & Roth, Learning a sparse representation

for object detection, 2002). The Vehicle detection and tracking contains two parts: first part is to successfully recognize the flying vehicle using part-based image representation method as described in paper (Agarwal, Awan, & Roth, Learning to detect objects in images via a sparse, part-based representation, 2004). Once the object is detected, its feature points and corresponding location is sent to KLT tracking algorithm to perform object tracking. If object get lost during tracking, detection algorithm will run again to detect new vehicles in the image.

1.2 PX4 Quadcopter

There is a trend in design and manufacture new types of multirotor aerial vehicles that are lightweight and more agile. Multirotor vehicle is ideal to both fly at wiled outdoor field and complex indoor environment, and it's so far the most successful type of aerial vehicle that can fly both at wield outdoor field and complex indoor environment.

AR.Drone is a quadrotor helicopter built by company Parrot. It has a cross-beam main frame, four electrical motor, two cameras, and a base house that protects and connects all the above components. It has proprietary flight management electrical board to run flight control software. In order to have greater access to the barebones hardware and add our custom flight control algorithm, we replaced the AR.Drone FMU with the open platform PX4.

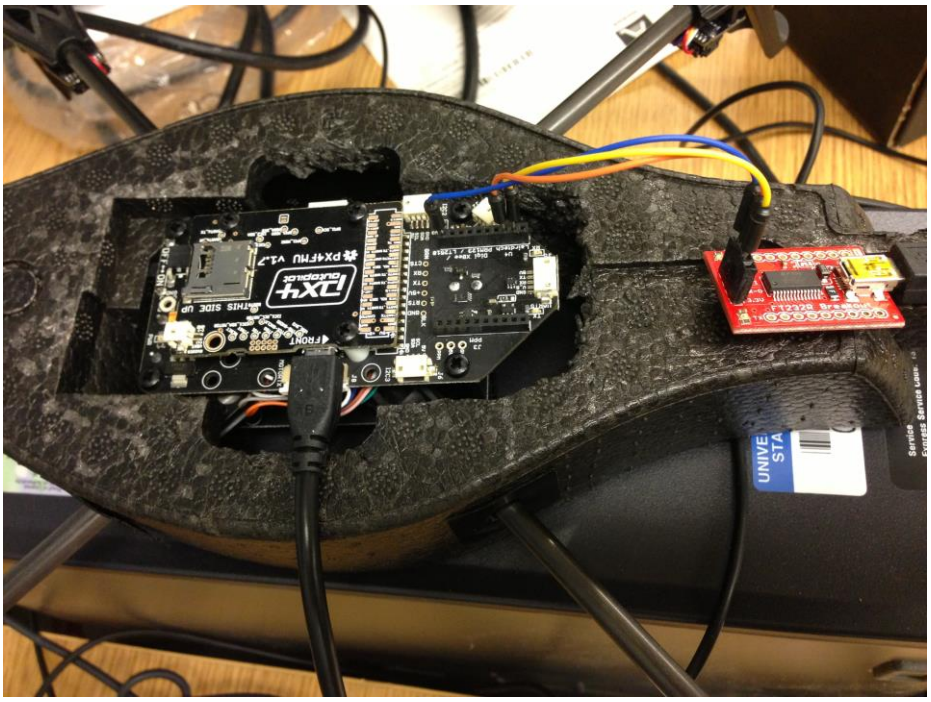


Figure 1: PX4 Flight Management Control Board

Figure 2: PX4 FMU with AR.Drone frame flying

The PX4 is an open-source, open-hardware platform aiming at providing high-end autopilot to the academic and industrial communities at low costs and high availability.

It is a complete software and hardware platform including an ARM based computer that can run multiple autopilot applications such as flight control stack and communication layers. It is a high-performance autopilot-on-module suitable for most autonomous vehicles. The key features and interfaces are listed below.

- 168 Mhz / 252 MIPS Cortex-M4F
- 182KB SRAM / 2014 KB Flash
- 4x UART, 2x I2C, 1x SPl, 1x CAN
- External magnetometer port
- RPM / RC Control input
- 8 GPIOs, 2 25mA high power, up to 4 PWM (servo out)

Figure 3： PX4 Flight Management Control Board

The autopilot modules runs a very power efficient real-time operating system (RTOS), which provides a POSIX-style environment. The benefit of running RTOS is to ensure processes with higher priority are guaranteed to have more computing resources to finish in time. This is more suitable for hard real-time applications such as flight control application than other operating systems.

1.3 Depth-Aided Vision Sensor

Camera with depth sensor open an entire new dimension for computer vision techniques. Microsoft releases Kinect depth vision sensor to aid their gaming console Xbox for human recognition and tracking. It is proven a huge success that Kinect enables a more human-interactive and dynamic gaming experience. Kinect is a motion-sensing device with depth sensing capability via infrared laser projector. It outputs video with frame rate of about 60 Hz. The RGB video stream has 8-bit 640x480 pixels with a Bayer color filter, while the depth sensing video stream is 11-bit. The horizontal minimum viewing distance is about 87cm, and the vertical one is about 63cm, resulting in a resolution of about 1.3mm per pixel.

Figure 4: Xbox One Kinect

Chapter 2: Object Recognition Via Sparse, Part-Based Representation

Since a successful object detection approach should be able to represent images in a manner that renders them invariant to intra-class variations, but at same time distinguishes images of the object class from other images. From paper (Agarwal, Awan, & Roth, Learning to detect objects in images via a sparse, part-based representation, 2004), Shivani Agarwal came up with a novel approach for learning to detect objects in images using a sparse, part-based representation. In this approach, the part-based representation is acquired automatically from a set of sample images of the object class of interest, thus capturing the variability in part appearances across the sample set. A classifier is then trained, using machine learning techniques, to distinguish between object and non-object images based on this representation; this learning stage further captures the variation in the part structure of object images across the training set. As shown in the experiments, the resulting algorithm is able to accurately detect objects in complex natural scenes.

This object recognition approach consists of five steps: data collection, vocabulary construction, image representation, learning a classifier, detection hypothesis and finally, testing.

2.1 Data Collection

The first stage is to collect image data that contains aerial vehicles, which is the sample data used for construct vehicle vocabulary as described in next section. Since aerial vehicles fly at six degree of freedom, they appear in the image with different poses, the sample data is collected with image taken from different angles of aerial vehicle flying at highly textured background. A subset of the sample is shown in the following image. They are are collected at Networked Autonomous Vehicle Laboratory at University of Illinois Urbana Champaign on August, 2014.

Figure 5: Sample datasets for training

Another data set is collect for training classifier. This data set composed of positive samples and negative samples. For each sample, the default sample size for PX4 quadcopter is 160×50, but can vary depends on the vehicle size. Positive samples contains target vehicle and negative samples contains pure background to ensure best training result. The sample collect rule is outlined at Algorithm 1.

Algorithm : VEHICLE DATA COLLECTION

Procedure DATA COLLECTION RULE

Initialize: $PositiveSet = empty, NegativeSet = empty$

Input: $sampleSet = \left\{ \begin{bmatrix} rgb_1 & \cdots & rgb_1 \\ \vdots & \ddots & \vdots \\ rgb_1 & \cdots & rgb_1 \end{bmatrix}, \dots, \begin{bmatrix} rgb_n & \cdots & rgb_n \\ \vdots & \ddots & \vdots \\ rgb_n & \cdots & rgb_n \end{bmatrix} \right\}$

For sample **in** sampleSet:

    If Vehicle exists, **Then:**

        PositiveSet ∪ sample

    Else:

        NegativeSet ∪ sample

End

Return PositiveSet, NegativeSet

Algorithm 1:Vehicle Data collection algorithm

2.2 Vocabulary Construction

The first step is building a vocabulary of parts that can be used to represent objects in the target class. To obtain an expressive representation for the object class of interest, we extract distinctive parts that are specifically belong to the object class. The parts are automatically selected based on extraction of interest points from the target object. A similar method has been used in (M, M, & P, 2000)

We applied Forstner interest operator to the vocabulary construction data set. This operator detects intersection points of lines and centers of circular patterns. Small image patches of size 13×13 then extracted from the interest points obtained. The goal

of extracting a large number of patches from different instances of the object class is to be able to reconstruct the new object instance.

After extracting the image patches, in order to facilitate learning and reduce overall computation overhead, it is necessary to abstract over the raw image patches by mapping similar patches to the same feature id (and distinct patches to different feature ids). Thus the bottom-up clustering procedure is used to accomplish this task. At first,

Algorithm: VEHICLE VOCABULARY CONSTRUCTION

Procedure VOCABULARY CONSTRUCTION RULE

Initialize: vocabulary = $[cluster_0]$, simThresh

Input: $img = \begin{bmatrix} rgb & \cdots & rgb \\ \vdots & \ddots & \vdots \\ rgb & \cdots & rgb \end{bmatrix} \in R^{m*n}$

Extract patches ← $interest\_operator(img)$

For patch **in** patches

Calculate similarity, idx ← $\max_{index}\{similar(patch, cluster) \mid \forall\ cluster \in vocabulary\}$

If similarity > simThresh: vocabulary → $\text{cluster}_{\text{idx}} \cup patch$

Else vocabulary ∪ patch

End For

Return vocabulary

Algorithm 2: Vehicle Vocabulary Construction Algorithm

each patch is assigned to a distinctive cluster, then each cluster which contains single patch start to successively merged to the clusters that have high similarity with each

other. During the merging procedure, the similarity between two clusters C_1 and C_2 is measured by the average similarity between their respective patches:

$$similarity(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{p_1 \in C_1} \sum_{p_2 \in C_2} similarity(p_1, p_2)$$

Where the similarity between two patches is measured by normalized correlation, allowing for small shifts of up to 2 pixels. Two clusters are merged if the similarity between them exceeds a similarity threshold (0.88 in the experiment). Through clustering method, originally 600 different patches are merged into 127 clusters, which is 4.7 times smaller. Although several clusters contain only one element, patches with high similarities are successfully grouped together.

2.3 Image Representation

After construct the vehicle vocabulary as described above, images can now represented by the vocabulary itself. This is accomplished by determining which vocab within vocabulary are present in the image, and representing it as a binary feature vector based on the detected vocab indices and the spatial relations that are observed among them.

Forstner operator is first applied to the image to extract image patches from the interest points detected. For each extracted patch, we perform a similarity-based indexing search into the vocabulary. Each patch is compared to the vocabulary parts using the same similarity measure used earlier for parts clustering when constructing

the vocabulary. If the similarity between extracted patch and vocabulary patch exceed a similarity threshold, the part in the image is then represented by the feature id corresponding the vocabulary part indices.

Finding out the vehicle consists of which parts does not give enough information to recover the target object. Since we assume aerial vehicles have rigid body, the geometric relationship between each pair of parts can give us a more detailed picture of what original target object might be. Several earlier approaches to recognition have used geometric constraints based on the distances and angles between object elements. For example, (T & W, 1985) models objects as polyhedral and, given input data from which object patches can be inferred, performs recognition by searching for interpretations of the surface patches whose inter-patch distances and angles are consistent with those between corresponding model faces. In this approach, we do not assume a known model for the target object class, but attempt to use the observed object parts in vocabulary constructing samples to learn a model to represent the object class. The distances and directions between parts are discretized into bins: in this experiment, distances are discretized into 5 bins based on their relative magnitude, and angles are discretized into 4 bins each covering angle of 90 degrees. Thus this gives us 20 possible distance-direction combinations between any two extracted parts. The image representation algorithm is shown in Algorithm 3.

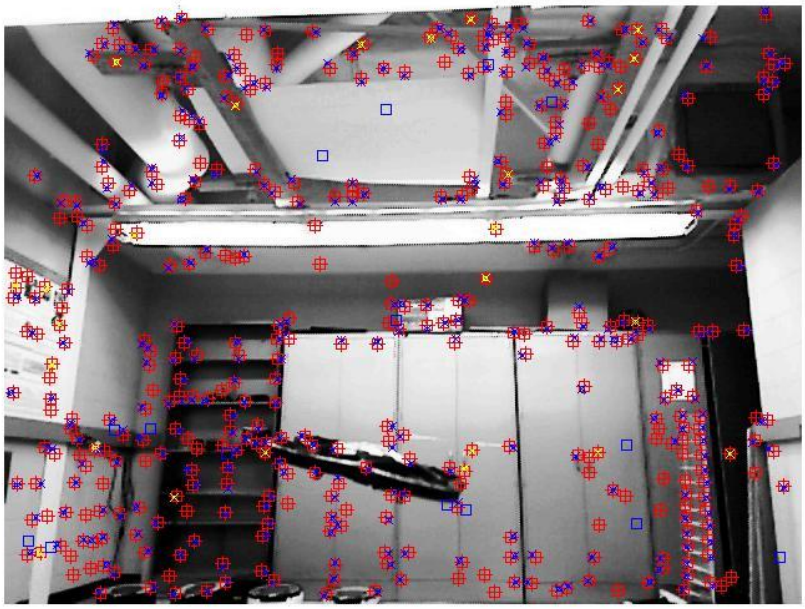


Figure 6: Fornster interest operator example

Algorithm: IMAGE REPRESENTATION

Procedure REPRESENT RULE

Input: $img = \begin{bmatrix} rgb & \cdots & rgb \\ \vdots & \ddots & \vdots \\ rgb & \cdots & rgb \end{bmatrix}$

Extract $patches \leftarrow interestOperator(img)$

For patch **in** patches

Match vocab $\leftarrow \underset{vocab}{\operatorname{argmax}}\{similarity(patch, vocab) \mid \forall\ vocab \in Vocabulary\}$

End

Convert features $\in \mathrm{N}^{\mathrm{N}} \xleftarrow[map\ f]{}$ {Distance(a, b), Angle(a, b) | ∀ a, b ∈ patches }

Return features

Algorithm 3: Image representation algorithm

The 160×50 training image is represented as a feature vector containing feature elements of two types:

i. $P_n^{(i)}$, *denoting the ith occurnace of a part of type n in the image* ($1 \leq n \leq 127$ *in this experiment, each n corresponds to a specific part cluster*)

ii. $R_m^{(j)}(P_{n1}, P_{n2})$, *denoting the jth occurance of relation* R_m *between a part of type* n_1 *and a part* n_2 *in the image* ($1 \leq \mathrm{m} \leq 20$ *in this experiment; each* R_m *corresponds to a particular distance direction combination*)

In the implementation, A part feature of $P_n^{(i)}$ is represented by a unique feature id, which is an integer determined as function of n and i. Similary, a relation feature of the form $R_m^{(j)}(P_{n1}, P_{n2})$ is assigned a unique feature id that is a function of m, n_1, n_2 and j.

$$\boldsymbol{Id_{feature} = max_{\{feature\ occurance\}} \cdot (n-1) + i} \qquad [\mathbf{1}]$$

$$\boldsymbol{Id_{relation} = max_{\{feature\ Id\}} + (n_1 - 1) \cdot C \cdot M \cdot max_{\{relation\ occurance\}} + (n_2 - 1) \cdot M \cdot max_{\{relation\ occurance\}} + (m-1) \cdot max_{\{relation\ occurance\}} + j} \qquad [\mathbf{2}]$$

Where C is number of clusters in the vocabulary (127 clusters in the experiment), M is total number of possible combination of relations (20 relations in the experiment). Thus the entire feature space has 759364 unique integer ids in total.

After applying the image representation algorithm on 40 positive samples, we found the average number of unique features exist in the image is 75. As we can see number of

features in one image is about one over then thousands of entire feature space, which leads to us picking SNoW classifier as described in next section.

2.4 Learning a classifier

Given a set of training images labeled as positive (object) or negative (non-object), each image is re-represented as a binary feature vector as described above. These feature vectors are then fed as input to a supervised learning algorithm that learns to classify an image as a member of non-member of the object class, with some associated confidence. The part-based representation captured by the feature vectors enables a relatively simple learning algorithm to learn a good classifier.

The SNoW[1] learning architecture is a sparse network of linear units over a common pre-defined or incrementally learned feature space. Nodes in the input layer of the network represent simple relations over the input and are being used as the input features. Each linear unit is called a target node and represents relations which are of interest over the input examples. Given a set of relations that may be of interest in the input example, each input example is mapped into a set of features which are active in it; this representation is presented to the input layer of SNoW and propagates to the target nodes. Target nodes are linked via weighted edges to some of the input features. Let $A_t = \{i_m , \dots , \}$ be the set of features that are active in an example and are linked to the target node t. Then the linear unit is active if and only if $\sum_{i \in A_i} w_i^t > \theta_t$, where w_i^t is

[1] Software for SNoW is available for free download at http://L2R.cs.uiuc.edu/~cogcomp/

the weight on the edge connecting the i^{th} feature to the target node t, and θ_t is its threshold. The learning policy is on-line and mistake-driven and several update rules can be used within SNoW. The most successful update rule is a variant of Littlestone's Winnow update rule, a multiplicative update rule tailored to the situation in which the set of input features is not known a priori, as in the infinite attribute model (Blum, 1992)

The Winnow update rule has, in addition to the threshold θ_t at the target t, two update parameters: a promotion parameter $\alpha > 1$ and a demotion parameter $0 < \beta < 1$. These are being used to update the current representation of the target t (the set of weights w_i^t) only when a mistake in prediction is made. Let $A_t = \{i_1, \dots, i_m\}$ be the set of active features that are linked to the target node t. If the algorithm predicts 0 (that is $\sum_{i \in A_i} w_i^t \leq \theta_t$) and the received label is 1, the active weights in the current example are promoted in a multiplicative fashion: $\forall i \in A_t, w_i^t \leftarrow \alpha \cdot w_i^t$. If the algorithm predicts 1 ($\sum_{i \in A_t} w_i^t > \theta_t$) and the received label is 0, the active weights in the current example is demoted: $\forall i \in A_t, w_i^t \leftarrow \beta \cdot w_i^t$. All other weights are unchanged. The key feature of the Winnow update rule is that the number of examples it requires to learn a linear function grows linearly with the number of relevant features and only logarithmically with the total number of features. This property seems crucial in domains in which the number of potential features is vast, but a relatively small number of them is relevant. Winnow is known to learn efficiently any linear threshold function and to be robust in the presence of various kinds of noise and in cases where no linear-threshold function can make perfect classification, and still maintain its abovementioned dependence on the number of total and relevant attributes (Littlestone, 1991).

Once target subnetworks have been learned and the network is being evaluated, a decision support mechanism is employed, which selects the dominant active target node in the SNoW unit via a winner-take-all mechanism to produce a final prediction. The algorithm is shown in Algorithm 4.

Algorithm: WINNOW UPDATE RULE

Procedure UPDATE RULE

Initialize: $\theta = \frac{2}{N}, W = \{\frac{1}{N}, \dots, \frac{1}{N}\}$

Input: $x = \{x_1, \dots, x_t\}, y = \{y_1, \dots y_t\}$

For $t = 1, \dots T$ **Do**

Get $x_t \in R^N$

Predict $\widehat{y_t} = sign(w_t \cdot x_t + \theta)$

Observe $y_t \in \{-1,1\}$

If $y_t = \widehat{y_t}, w_{t+1} = w_t$

Else $w_{t+1,i} = w_{t,i} e^{\eta y_t x_{t,i}}$

End For

Return θ, W

Algorithm 4: Winnow update algorithm

The SNoW classifier can run in three major modes: training, testing and evaluation. Training takes a set of labeled examples and generates a network which can be used to make predictions on future examples. Thus training must always be performed before testing or evaluations can be performed. Testing is performed on a set of (labeled or

unlabeled) examples contained in a file. The results of the test are written to a file. Evaluation is used to make a prediction based on a labeled or unlabeled example supplied on the command line.

In order to use SNoW classifier to learn the target objects, you need to take following steps:

1) *Collect sample vehicle images as described in section 2.1 data collection for vocabulary construction and classifier training*
2) *Use vocabulary construction data samples to construct vehicle vocabulary, the vocabulary should include:*
    a. *clusterPool: contains all clusters, each cluster contains the list of patch indices that belongs to this cluster*
    b. *imgHeight: height of sample image*
    c. *imgWidth: width of sample image*
    d. *patchPool: contains all image patches, which are* 15×15 *pixel matrixes*
    e. *patchSize: the size of each patch*
3) *Use the vocabulary to represent all sample images in training data set and store it in a text training file. For positive training data, label 1 before the representation; for negative training data, label 0 before the representation. See example training file in* Figure 7

After training file is created, train the SNoW classifier using the training file as described in Appendix B: SNoW Classifier User Guide. The classifier training algorithm is shown in Algorithm 5

Figure 7: Example of training-file

Algorithm: CLASSIFIER TRAINING

Procedure TRAINING RULE

Initialize: $trainingFile = empty,\ Classifier$

Input: $samples = \{sample_1, \dots, sample_n\},\ where\ sample_i \in R^{m*n}$

For $sample_i$ **in** samples

Represent $features_i, \hat{y}_i \xleftarrow[represent]{} sample_i$

Record $trainingFile + features_i, \hat{y}_i$

End For

Train $trainingFile \xrightarrow[feed]{} Classifier$ using Winnow Update Rule

Return Classifier

Algorithm 5: Classifier training algorithm

2.5 Detection Hypothesis using the Learned Classifier

The final stage consists of using the learned classifier to form a reliable detector. Having learned a classifier that can classify 160×50 images as positive or negative, aerial vehicles can be detected by sliding a 160×50 window over the image and by classifying each window we can learn where the vehicle locate at image. However, it is clear that near an object, several windows might classify the same object as positive, giving rise to duplicate detections to a single object in the scene.

We introduced the notion of a classifier activation map, which can be generated from any classifier that produce a binary classification output along with a confidence number. Negatively classified windows are mapped to a zero activation value, and positively classified ones are mapped to the activation value produced by the classifier. This produces a map with high activation values at points where the classifier has high confidence in its positive classification. Then user Neighbor Suppression Algorithm, which searches for activation peaks in a rectangular neighborhood, defined by two parameters $heightBound$ and $widthBound$, so that a point (i_0, j_0) is considered to be an object location if:

i. $activation(i_0, j_0) \geq activation(i, j) for\ all\ (i, j) \in N, where\ N = \{(i, j): |i - i_0| \leq heightBound, |j - j_0| \leq widthBound\}$

ii. $no\ other\ point\ in\ N\ has\ already\ been\ declared\ an\ object\ location$

The neighborhood size determines the extent of overlap that is allowed between detected windows, and can be chosen appropriately depending on the object class and

window size. In this experiment, we use $heightBound$ = 25 pixels, and $widthBound$ = 60 pixels. The vehicle detection and recognizances algorithm is shown in

There is trade-off between the number of correct detections and number of false detections. An activation threshold is introduced in the algorithm to determine where to lie on the trade-off curve. All activations in the activation map fall below the threshold are set to zero. Lowering the threshold increases the correct detections but also increases the false positives, and also decrease the detection speed; on the other hand raising the threshold has the opposite effect.

Algorithm: Vehicle Detection and Recognizances

Procedure UPDATE RULE

Initialize: Vehicles = empty, Classifier, y_θ

While true, **do:**

Get $img = \begin{bmatrix} rgb & \cdots & rgb \\ \vdots & \ddots & \vdots \\ rgb & \cdots & rgb \end{bmatrix}, depth = \begin{bmatrix} depth & \cdots & depth \\ \vdots & \ddots & \vdots \\ depth & \cdots & depth \end{bmatrix} \in R^{M*N}$

Represent $[img, depth] \xrightarrow[map\ f]{} features \in R^N$

Classify $\hat{y}_{1,\dots n} = Classifier.predict\{features\} \in [0,1]^n$

Eliminate Duplication $\hat{y}_{1,\dots n} \xrightarrow[\boldsymbol{suppression}]{} \hat{y}_{1,\dots m}\ m \leq n$

Detect $y = \hat{y}_{1,\dots m} \geq y_\theta$

End While

Track Vehicles

Algorithm 6: Vehicle detection and recognizances algorithm

Chapter 3: Depth-Assistant Real-Time Optimization

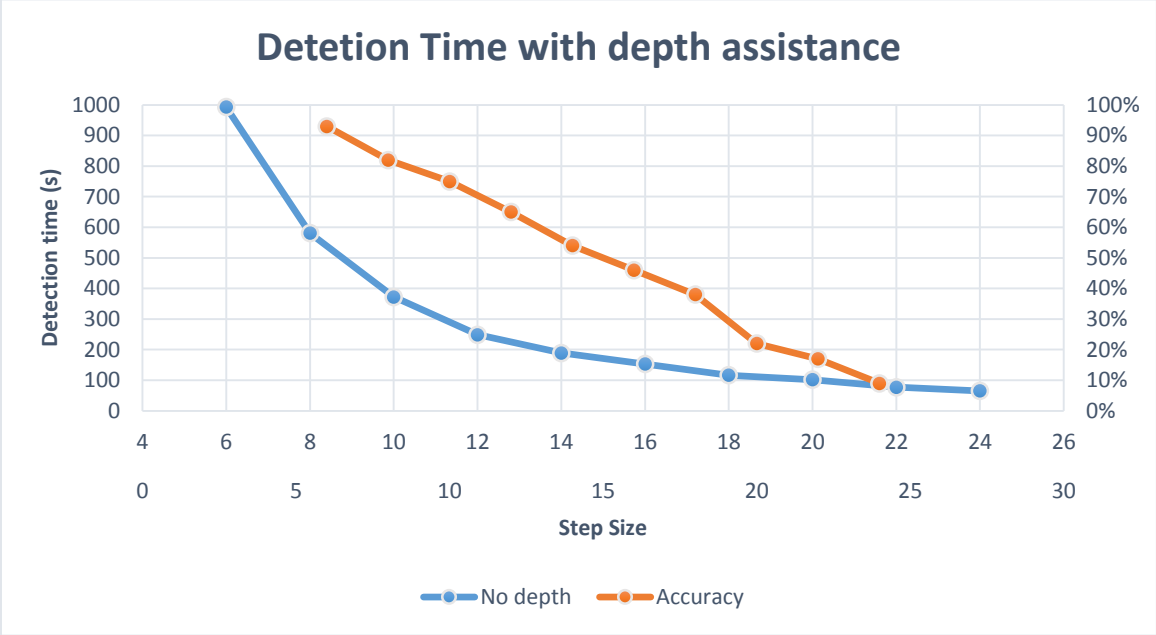
The object recognition via sparse, part-based representation can detect target object class in the given image frame. When this algorithm is implemented in Matlab on Intel i7 computing platform, it takes 16 minutes to accomplish detection task on one 640×480 image. This is not acceptable when our goal is to detect aerial vehicle in real-time. In order to significantly speed up the entire process, we proposed a novel approach called Depth Aided Image Pre-Processing, which utilize the advantage of depth vision sensor combined with normal RGB sensor to pre process the image to find global gist of where vehicle might located. As shown in both theoretical and experimentally below, this speed up the detection process by more than 1000 times faster without sacrificing the detection accuracy. Besides we also proposed several optimization on computation heavy procedures to further increase computation speed.

3.1 Depth Aided Image Pre-Processing

The most computation intensive task is to slide window over entire image to locate where vehicle might be. For instance, given the image size 640×480, and sliding windows size 160×50, step size 1, the window slides over 206400 possible vehicle locations. On each location, the windows need to be represented to binary vector and feed to classifier for classification. This is $O(wh)$ operation where w is image width and h is image height. If we want to keep the entire process down to 1 seconds, the window

at each location needs to be processed in less than 4.8 microseconds, which is impossible to achieve with moderate computing platform.

One way to decrease increase window sliding efficiency is to increase step size. The step size is how much pixels window slides each time. The bigger step size, window slides faster and less possible vehicle locations are visited. The relationship between step size and overall computation complexity is $O(\frac{wh}{2^{N-1}})$, where N is the step size. As we can see the computation complexity decreases exponential with increase of step size. However, this comes the sacrifice of accuracy of vehicle detection. Because each time window slides, the pixels it missed might just be the exact vehicle location. We plot the graph of the detection time versus step size as shown in Plot 1: when step size is bigger than 20, the detection time successfully decreases under 100 seconds, but the detection accuracy falls just below 10%.



Plot 1: Detection time and detection accuray with depth assistance

The major breakthrough to significantly decrease the runtime of this process is to introduce depth-aided image preprocessing. Since vehicle flying in the background has depth different from the depth of background, if we use edge detection method to find edges that has large depth gradient, we will be able to distinguish and locate objects that are closer to the camera before we classify which objects are the flying vehicles as shown in figure 1-3. In each image we combine depth data and RGB data and we usually detect 2-10 possible vehicle locations, the total number of sub-image we need to classify is thus O(1) which is constant time complexity. The depth-aided image preprocessing algorithm is shown in Algorithm 7.

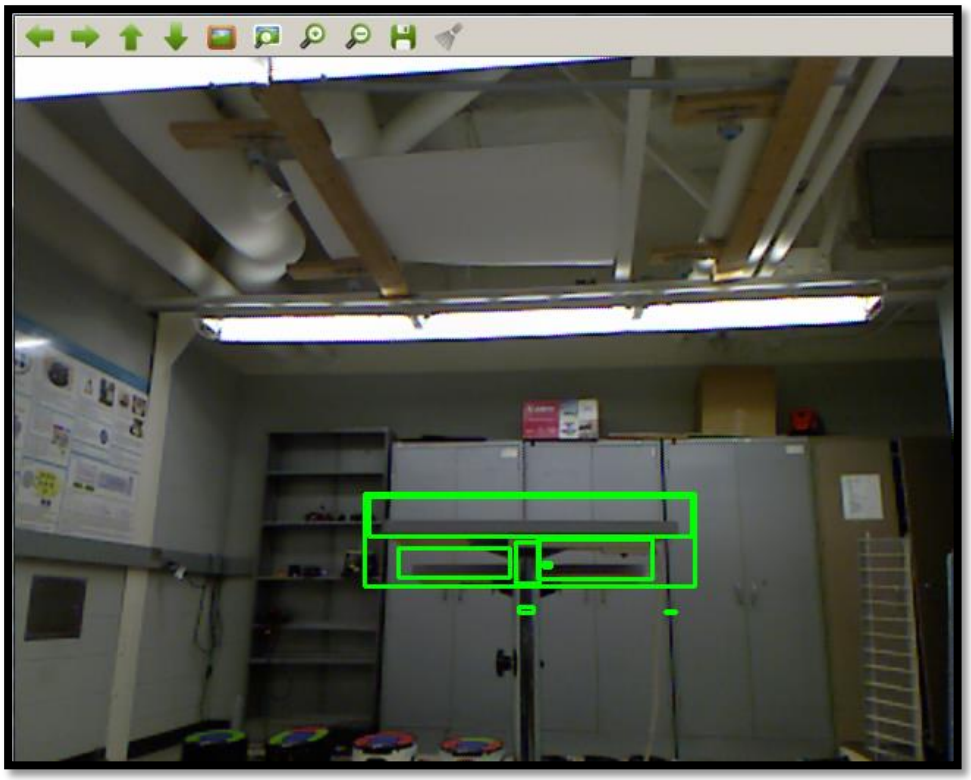


Figure 8: Contour selection algorithm to detect objects different from the background

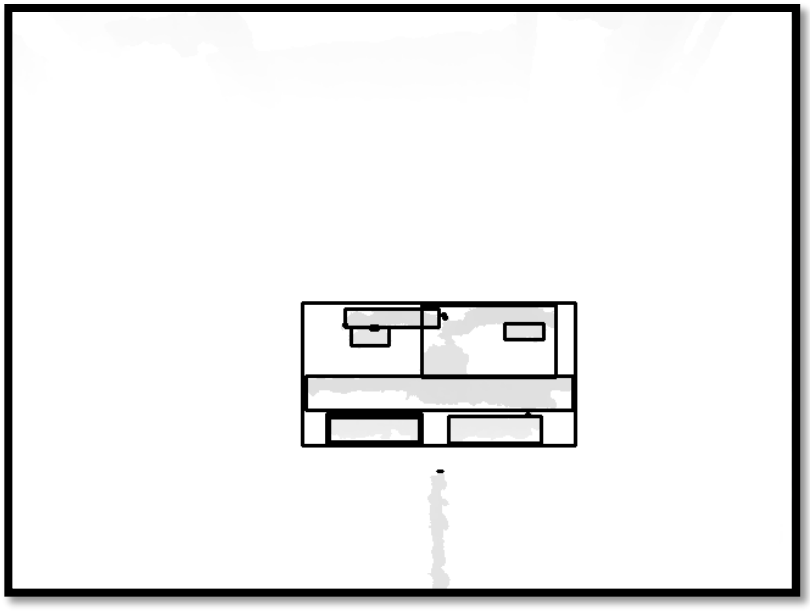


Figure 9: Contour selection algorithm to detect objects different from the background

Algorithm: DEPTH IMAGE PREPROCESSING

Procedure PROCESSING RULE

Initialize: $global\_locations = empty$

Input: $img = \begin{bmatrix} rgb & \cdots & rgb \\ \vdots & \ddots & \vdots \\ rgb & \cdots & rgb \end{bmatrix}, depth = \begin{bmatrix} depth & \cdots & depth \\ \vdots & \ddots & \vdots \\ depth & \cdots & depth \end{bmatrix}$

Detect $edges = \nabla depth > \theta, \nabla f = \frac{\partial f}{\partial x}\hat{x} + \frac{\partial f}{\partial y}\hat{y}$

Bounding $rectangles \xleftarrow[Min\ area\ enclosing\ rectangles]{} edges$

For each rectangle**, Do**

Centroid $\bar{x} = \frac{1}{A}\int_a^b x[f(x) - g(x)]dx$

$\bar{y} = \frac{1}{A}\int_a^b \left[\frac{f(x)-g(x)}{2}\right][f(x) - g(x)]dx$

$global\_locations \cup (\bar{x}, \bar{y})$

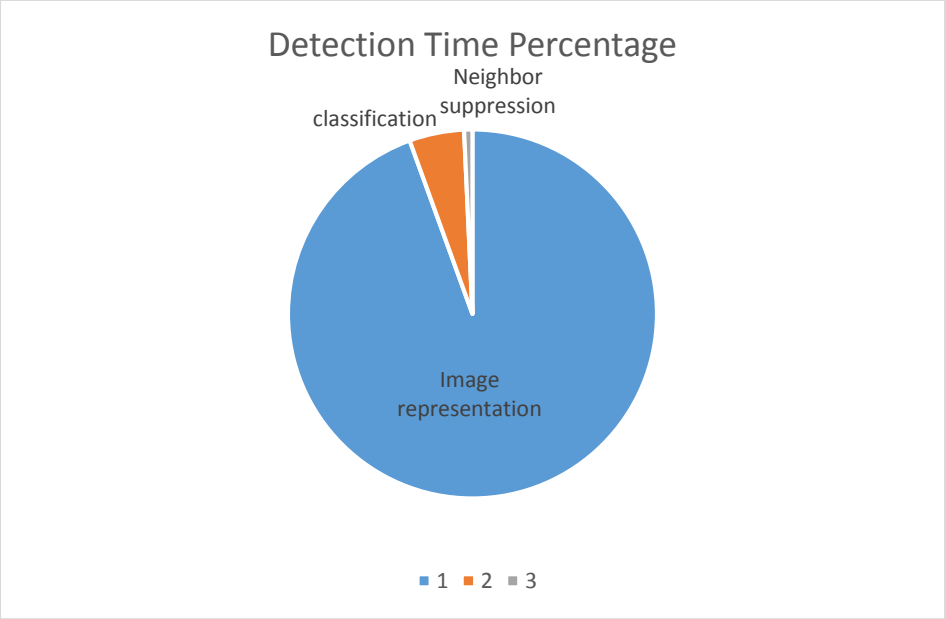
End For

Return global_locations

Algorithm 7: Depth Image Preprocessing Algorithm

3.2 Optimized Vocabulary Search

After we run the detection algorithm several times, the image representation time consistently takes more than 80% of overall detection time as shown in Plot 2. After we analysis each function runtime inside image representation algorithm, we found one of another major computation bottleneck is the procedure of looking for the cluster from the vocabulary that has the highest similarity with the given image patch. We know the optimal size of cluster is about 127, thus each image patch needs to compare 127 clusters to find the best match, and the average number of image patches is about 89 in our experiments, thus it is important to make this process more efficient than $O(MN)$, where M is number of patches and N is size of vocabulary. One technique we used is memorization. Since each sub-image contains multiple patches, and two subsequent sub-images overlap with each other, there are common image patches exist at both sub-images as shown in Figure 10. If feature points tend to cluster, we found this algorithm increases efficiency by memorizing the common features' similarity cluster index so that there is no need to calculate again when the feature point appears in next sliding sub-image. In order to further decrease algorithm run time, we use lazy match instead of full comparison between image patch and all clusters in the vocabulary. Lazy match means when looking for the most similarity clusters compared to the image patch, the search will immediately stop when the similarity is larger than similarity threshold instead of doing full search for the optimal similarity. Since we conduct random search inside subset of vocabulary, the amortized run time would be $O(M)$, which is more efficient than original run time $O(MN)$. The algorithm is presented in Algorithm 8.



Plot 2: Detection Time Percentage

Algorithm: Vocabulary Lazy Match with Memorization

Procedure MEMORIZATION OPTIMIZATION RULE

Initialize: Vocabulary, SimThresh, DynMem= $\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \in R^{m*n}$

Input: img= $\begin{bmatrix} rgb & \cdots & rgb \\ \vdots & \ddots & \vdots \\ rgb & \cdots & rgb \end{bmatrix}$, features= $[point_1 \ \ldots point_n]$

For point **in** features

Retrive maxSim **from** DynMem

Calculate $similarity = \max_i \{similar\{cluster_i, patch\}\} \ \forall \ cluster_i \ \in Vocabulary$

Break If $similarity \leq SimThresh$

Store $similarity \ at \ DynMem[point.x, point.y]$

End For

Algorithm 8: Vocabulary Lazy Match with Memorization

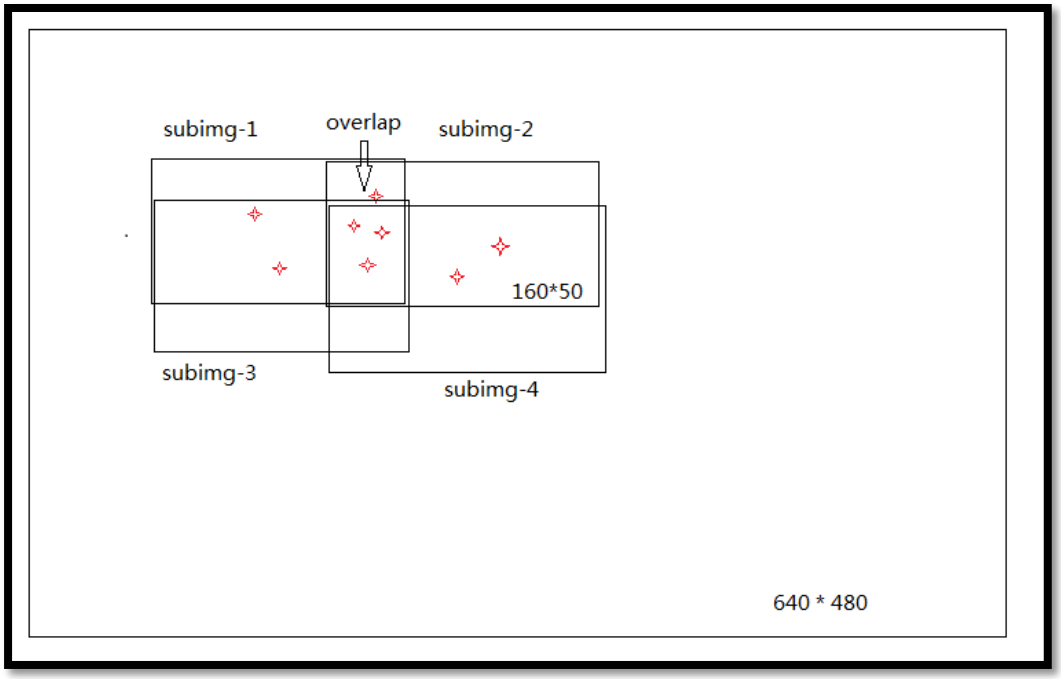


Figure 10: Sub image overlapping

3.3 Linear Neighbor Suppression

With the depth aided image preprocessing, number of sub-image needs to classify signicantly decrease to order of O(1), and we found the original Neighbor Suppression algorithm described in paper (Agarwal, Awan, & Roth, 2004) incur large overhead with the usage of two dimensional matrix to store all activation records. Since without depth assistance, number of activation records is about 4103, by placing them in the two dimensional matrix it's easier to determine and suppress the neighbors. Now the number of activation records is only 2-10, and it would be inefficient to place tiny amount of values into 640 × 480 matrix. Thus the Linear Neighbor Suppression algorithm we introduce eliminate the need to create two dimensional matrix and use only one dimensional array to store all activation records. Given the number of

activation records is small, for each activation record, we iterate through all other records to find its neighbor and compare the two values, and we suppress all neighbors' values if it's activation is higher than its neighbors. Algorithm is presented in Algorithm 9.

Algorithm: LINEAR NEIGHBOR SUPPRESSION RULE

Procedure SUPPRESSION RULE

Initialize: detections = []

Input: Activations= $[acti_1, \cdots, acti_n] \in [0,1]^N$, Coordinates= $[coor_1, \dots, coor_n]$

For $acti_i$ **in** Activations, $coor_i$ **in** Coordinates, **Do**

    For j = (i+1) **to** n, **Do**

        Retrieve $Activations_j, Coordinates_j$

        Suppress $acti_m$ **If** $Activations_j \geq acti_m \; \forall \; acti_m \in \{Neighbor\ of\ Coordinates_j\}$

    End For

End For

Detect detections ∪ $Activations_i \neq 0$

Return detections

Algorithm 9: Linear neighbor suppression algorithm

3.4 Expectations

With the above optimizations, we expect to see shear performance increase due to the novel depth image preprocessing technique, also marginal performance increase with optimized vocabulary search and linear neighbor suppression algorithms.

Chapter 4: Vehicle Tracking

After aerial vehicle is recognized and localized in the given image, its location needs to get continuous update in the following images. The indoor aerial vehicle tracking is difficult because aerial vehicle fly with different poses and subject to fast background change. The current mature and reliable way is to use Vicon Vision System. It is an expensive vision sensor set that use feature objects attached on the target to track the target. The problem with this approach is its expensive and labor cost to set up entire system, also without additional feature objects attached, it is unable to track target objects.

Since we have already developed an algorithm that detects target object with real-time performance, we can repeatedly recognize and localize vehicle in consecutive images to get vehicle location update. However, this approach requires powerful computing platform to compensate recognize computation overhead, and this approach might cause inconsistency in tracking due to imperfect detection accuracy.

Our approach is to use KLT Feature Tracker algorithm to track the detected vehicle. When it gets lost by the tracker, the detector will kick in again to recognize aerial vehicles inside image.

4.1 Kanade-Lucas-Tomasi Feature Tracker

The original image alignment algorithm was the Lucas-Kanade algorithm, and the goal of Lucas-Kanade is to align a template image $\boldsymbol{T}(\boldsymbol{x})$ to an input image $\boldsymbol{I}(\boldsymbol{x})$,

where $\mathbf{x} = (\mathrm{x}, \mathrm{y})^{\mathrm{T}}$ is a column vector containing the pixel coordinates. If the Lucas-Kanade algorithm is being used to compute optical flow or to track an image patch from $t = 1$ to time $t = 2$, the template $\boldsymbol{T}(\boldsymbol{x})$ is an extracted sub-region of the image at $t = 1$ and $\boldsymbol{I}(\boldsymbol{x})$ is the image at $t = 2$.

Let $\mathbf{W}(\mathbf{x}; \mathbf{p})$ denote the parameterized set of allowed warps, where $\mathrm{p} = (\mathrm{p}_1, \ldots \mathrm{p}_\mathrm{n})^{\mathrm{T}}$ is a vector of parameters. The warp W(x; p) takes the pixel x in the coordinate frame of the image I. If we are computing optical flow, for example, the warps W(x; p) might be the translations:

$$\mathrm{W}(\mathrm{x}; \mathrm{p}) = \begin{pmatrix} x + p_1 \\ y + p_2 \end{pmatrix} \qquad [3]$$

where the vector of parameters $\mathrm{p} = (\mathrm{p}_1, \mathrm{p}_2)^{\mathrm{T}}$ is then the optical flow. If we are tracking a larger image patch moving in 3D we may instead consider the set of affine warps:

$$\mathrm{W}(\mathrm{x}; \mathrm{p}) = \begin{pmatrix} 1 + p_1 & p_3 & p_5 \\ p_2 & 1 + p_4 & p_6 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad [4]$$

Where there are 6 parameters $\mathrm{p} = (\mathrm{p}_1, \mathrm{p}_2, \mathrm{p}_3, \mathrm{p}_4, \mathrm{p}_5, \mathrm{p}_6)^{\mathrm{T}}$ as, for example, was done in (Bergen, Anandan, Hanna, & Hingorani, 1992). In general, the number of parameters n may be arbitrarily large and W(x; p) can be arbitrarily complex.

The goal of the Lucas-Kanade algorithm is to minimize the sum of squared error between two images, the template T and the image I warped back onto the coordinate frame of the template:

$$\sum_x \left[I\big(W(x; p)\big) - T(x) \right]^2 \qquad [5]$$

Warping I back to compute I(W(x; p)) requires interpolating the image I at the sub-pixel locations W(x; p). The minimization of the expression in Eq. (3) is performed with respect to p and the sum is performend over all of the pixels x in the template image T(x). Minimizing the expression in Eq. (1) is a non-linear optimization task even if W(x; p) is linear in p because the pixel values I(x) are, in general, non-linear in x. In fact, the pixel values I(x) are essentially un-related to the pixel coordinates x. To optimize the expression in Eq. (3), the Lucas-Kanade algorithm assumes that a current estimate of p is known and then iteratively solves for increments to the parameters Δp; i.e. the following expression is approximately minimized:

$$\sum_{x}[I(W(x;p+\Delta p))-T(x)]^2 \qquad [6]$$

With respect to Δp, and then the parameters are updated:

$$p \leftarrow p + \Delta p \qquad [7]$$

These two steps are iterated until the estimates of the parameters p converge. Typically the test for convergence is whether some norm of the vector Δp is below a threshold ϵ; i.e. $||\Delta p|| \leq \epsilon$.

4.2 Feature Extraction

The KLT algorithm needs to track interest points. As we did in the image representation step, the interest points are extracted by Forenster operator, and can

be stored in a matrix. After the vehicle is detected and localized, the interest points inside detected window 160×50 are stored and used directly for feature tracking.

4.3 Vehicle Tracking

The KLT feature tracking algorithm continue to track features until features get lost. Then the object recognition algorithm start working again as described by the following logic diagram.

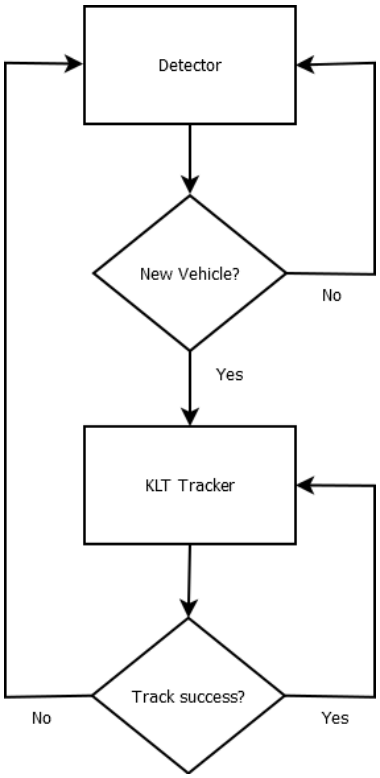


Figure 11: Vehicle detect and track logical diagram

Chapter 5: Implementation

The HotDec Testbed at Networked Autonomous Vehicle Laboratory has mature software architecture to conduct various types of experiments on hovering vehicles like Hovercraft. With the addition of new real-time aerial vehicle recognition and tracking system, the HotDec Testbed will be able to conduct experiments on indoor aerial vehicles more successfully. Here I introduce how I implemented the real-time indoor aerial vehicle recognition system and, on top of it, how it got integrated into the existing HotDec Testbed infrastructure and make them work seamlessly with each other.

5.1 Software Architecture

The software architecture of the system mainly consists of three components: kinect vision server, machine learning server, and aerial vehicle ground control station. The Kinect vision server runs depth image processing service that first use depth image pre-processing to calculate image global gist, then represent image by constructed vehicle vocabulary to produce binary feature vector. The binary feature vector then fed to the machine learning server for classification. The service diagram is shown in Figure 12.

Figure 12: The kinect Server service diagram

The machine learning server runs SNoW classifier with server mode enabled, which means it can accept request from local machine as well as from any other machine in the lab. The SNoW classifier can be trained to detect multiple object classes.

The aerial vehicle ground control station is the central command center for PX4 quadcopter. It acts as a relay between various services in the lab and the aerial vehicle. For example, it subscribes location messages from Kinect vision server and delivers to the PX4 quadcopter for location feedback. It also collects information from various services and process information and send final command to the aerial vehicles.

5.2 Hardware Acceleration

Although it is obvious to use powerful GPU to do graphics computation, sometimes GPU tend to lose its computational efficiency or even worse than CPU when a target algorithm has complicated decision flows depending on the current evaluation of input data. Since several most complex computations including corner detection, Hessian

inverse and vocabulary searching all require quite lot logical computation and data transfer, we adopt a CPU first GPU second hybrid approach that emphasize optimizing CPU performance and make use of GPU as secondary option. This approach also aligns our goal to create best real-time performance on moderate computing hardware.

In order to best utilize CPU power, we integrate a library of highly optimized algorithm building blocks for media and data applications: the Intel Integrated Performance Primitives (IPP). Intel IPP is a cross-platform API suit; it provides API for many computational intensive tasks such as image process, audio decoding and encryption etc. it supports various data structures and at meantime minimalize their memory consumption.

Originally, most computer vision tasks are performed by OpenCV library. OpenCV is an open sourced comprehensive computation vision library that provides extensive computer vision APIs but delivers moderate performance. In the KLT tracking algorithm, IPP image warping and feature selection APIs are used to boost calculation speed, and it gives us 2X speedup vs OpenCV library.

5.3 Laboratory Setup

Previous graduate student Richard Otap created a new service call auto discovery service that allows machines with different services automatically find each other in the same network and use each other's services. He also proposed a messaging service called zero message queue. This is now the central messaging protocol between vehicles, machines and

all other services. The PX4 quadcopter, aerial vehicle ground station and machine learning server all utilizes this message service, enables tight integration with existing HotDec infrastructure and allows easy scalability in future new addition services.

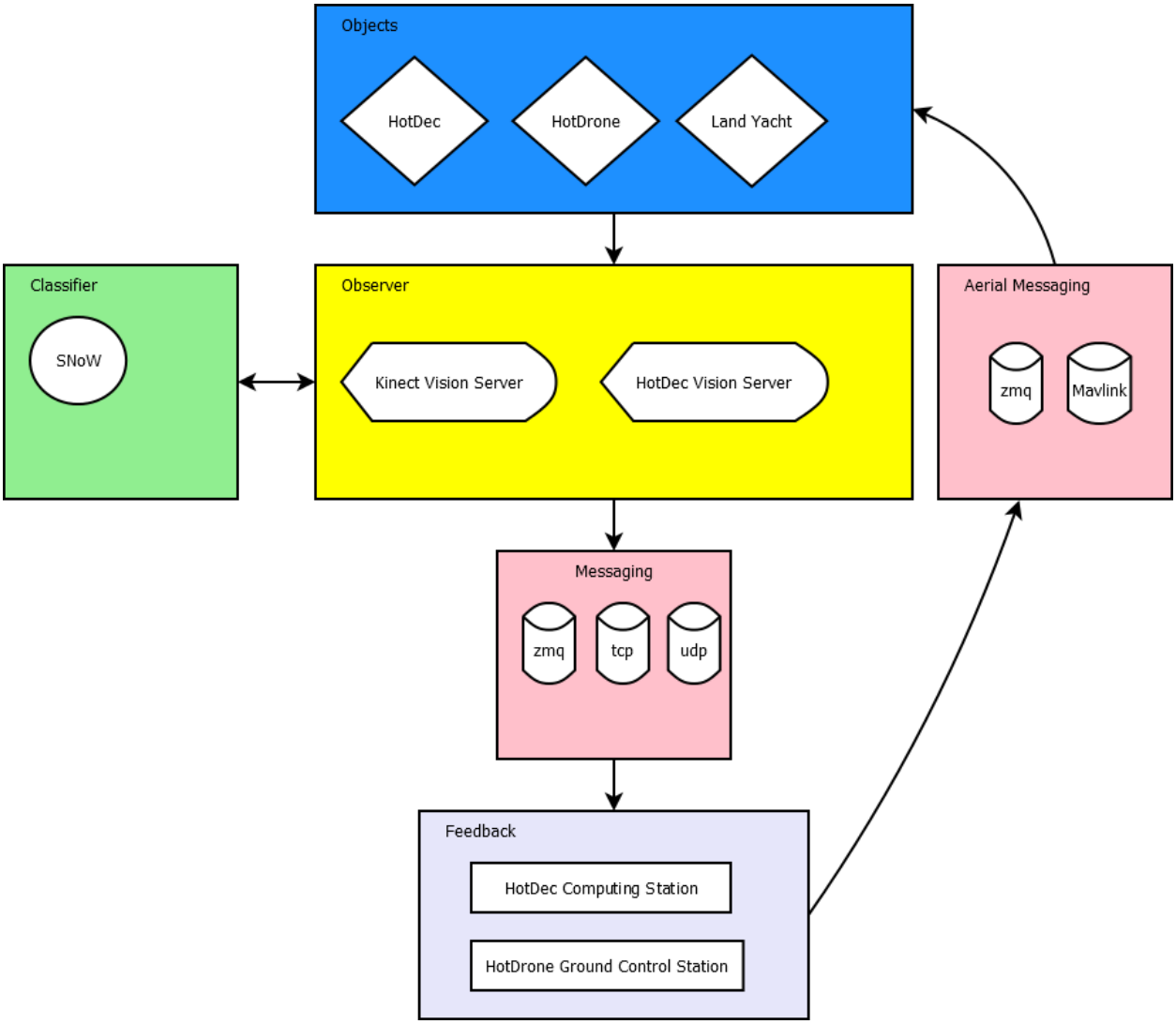


Figure 13: Aerial Vehicle Integrated HotDec Testbed

Since the Kinect Vision Server runs on single machine, it subjects to single point of failure that could ruin the entire service. In order to be able to recover efficiently in the future, backup an Ubuntu image periodically is strongly recommended.

However, in case of necessity of a total system reinstall, I wrote out a software installation procedures that include step by step tutorial of how to run the aerial vehicle real-time detection and tracking software on a new set-up machine in references.

Chapter 6: Experimental Evaluation

6.1 Data Sets

We collect 250 RGB and depth images contains drone as positive samples, and 250 RGB and depth images does not contain drone as negative samples. The size of drones are roughly the same but they appear in different poses and locations in the images. The images were all taken with in Networked Autonomous Vehicle Laboratory at Mechanical Engineering Lab Building, and acquired by Kinect 1.0 sensor with highly textured backgrounds. The sample images are shown below.

Figure 14: PX4 drone flying in lab

6.2 Evaluation Criteria

In the positive samples, we determine the location of vehicles manually in 160 x 50 window containing the drone. For a location output by detector to be evaluated as a

correct detection, we require it to lie within an ellipse of a size centered at the true location as described in paper [2].

$$\frac{|\mathrm{x} - \mathrm{x}^*|^2}{\alpha_{width}^2} + \frac{|y - y^*|^2}{\alpha_{height}^2} \leq 1 \qquad [7]$$

Where α_{width} and α_{height} determine the size of allowed ellipse and x, y is the true location of the vehicle and x^*, y^* is the location output by the detector. We allow axes of the ellipse to be 30% of the object size along each dimension. In order to conduct speed comparison between algorithms, we first run all the algorithms inside Matlab with same hardware configuration then we run the same algorithm with C++ implementation to achieve best performance. We use Matlab tic, toc throughout Matlab and ctime in C++ as timing mechanisms.

6.3 Performance Measures

To measure the performance of object detection, we want to maximize the number of true positive detections and minimize false positive detections. We define the number of true positive detections as TP, number of false positive detections as FP, total number of positive samples as nP and total number of negative samples as nN, thus:

$$\text{True positive rate} = \frac{\mathrm{TP}}{nP} \qquad [8]$$

$$\text{False positive rate} = \frac{\mathrm{FP}}{nN} \qquad [9]$$

Then as a detection system we define two more terms to capture how many of the objects it detects and how often the detections it makes are false.

$$\text{Recall} = \frac{\text{TP}}{nP} \qquad [10]$$

$$1 - \text{Precision} = 1 - \frac{\text{TP}}{TP + FP} \qquad [11]$$

Where Recall is the proportion of the objects detected and 1-Precision is the number of false detections relative to the total number of detections made by system as introduced in paper (Agarwal, Awan, & Roth, 2004).

6.4 Experimental Procedure

The experiment is conducted at Networked Autonomous Vehicle Laboratory. In order to run algorithm efficiently, a series of images of PX4 quadcopter flying in the laboratory is recorded by Kinect vision sensor, along with their corresponding depth images. The total image count is 250, then for each image, if the vehicle is inside image, the location of vehicle is manually located; otherwise the location is labelled as at pixel [0,0]. Then first run the aerial vehicle recognition algorithm only and without real-time optimization techniques to calculate the vehicle detection accuracy. Several parameters are tuned and compared in order to get best accuracy while maintain computation overhead low.

Then run the vehicle recognition algorithm with the depth assistant real-time optimization techniques to see the performance boost and compare the results with the one without optimization.

After vehicle recognition algorithm parameters are tuned towards the best performance measurement, then kick in the tracking algorithm to examine the tracking performance.

Finally run aerial vehicle detection and tracking algorithm together to perform the laboratory demo.

Chapter 7: Results

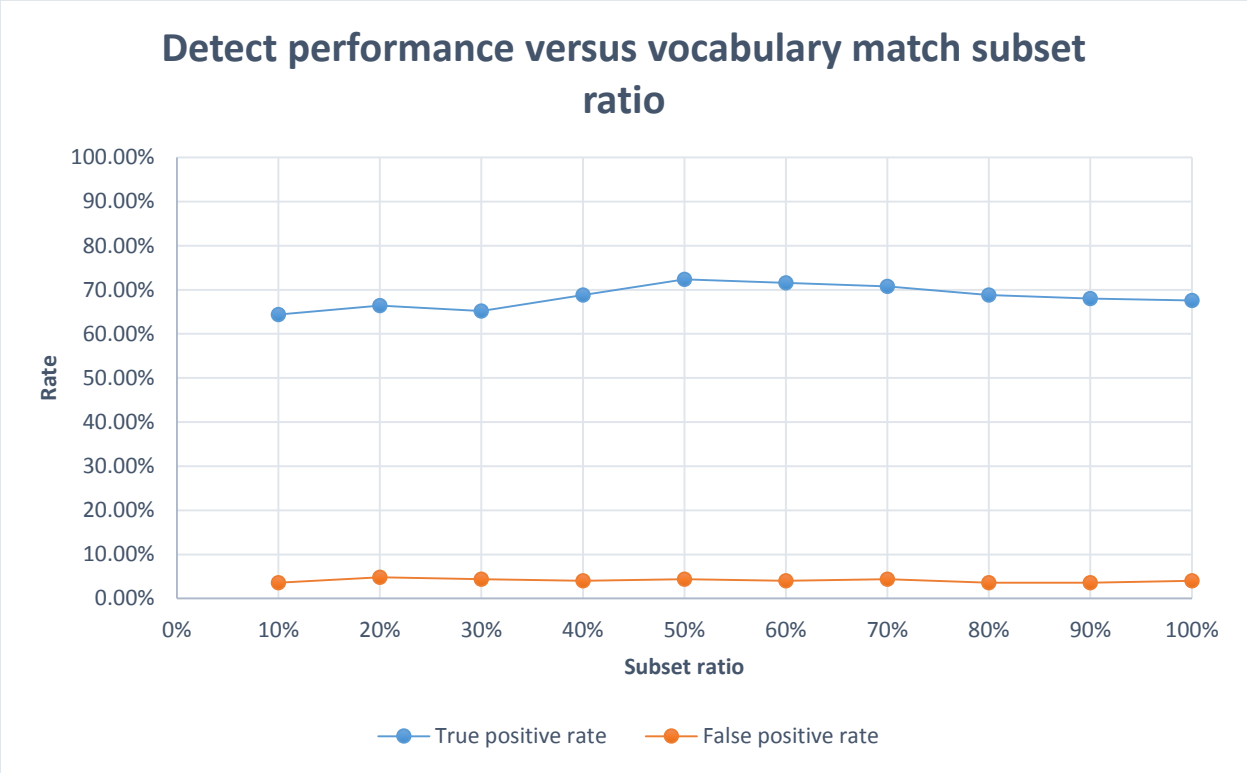
7.1 Vocabulary Matching Methods Comparison

In optimized vocabulary search, we proposed search only partial of entire vocabulary to reduce search time. The subset ratio controls how much of the vocabulary is being searched. As we can see from Table 1, with the subset ratio vocabulary matching algorithm used varied from 10% to 100%, the true positive rate vary from lowest 64.4% at subset ratio equals 0.1, to highest 72.4% at subset ratio 0.5. At meantime, with increasing value of subset ratio, the amount of computation time required increases almost linearly as shown in Plot 4. For larger subset ratio, larger subset of patches in one cluster need to be compared with and thus increase computation overhead. Since we want to increase algorithm speed for real-time purpose, we weight the importance of speed increase more than marginal detect performance decrease as shown in Plot 3. After rigorous tuning, we believe when subset ratio equals 0.2, the algorithm gives good performance while keep computation overhead low. From Plot 5, for different steps sizes when iterating over image, there is marginal speed increase over original vocabulary matching method as expected. When step size is small, the speed increase is larger because more common feature points resides at multiple sub images thus memorization can save more computing time.

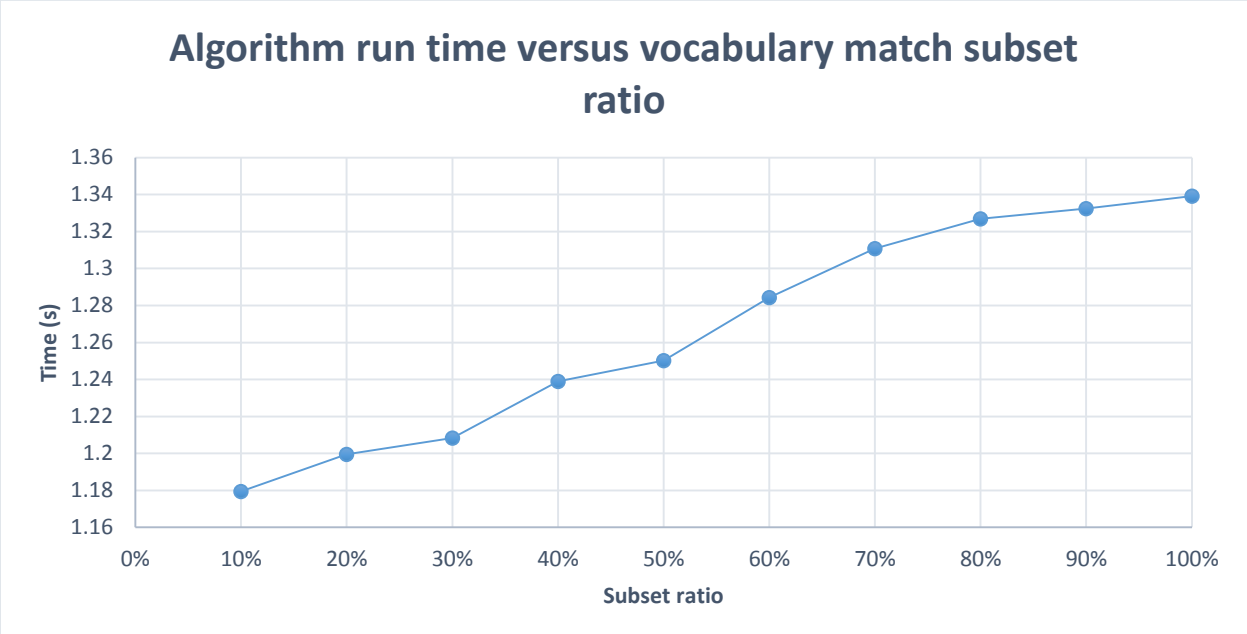
Figure 15: Aerial Vehicle Detected and Tracked

Table 1: Detect prevision versus subset ratio in vocabulary search

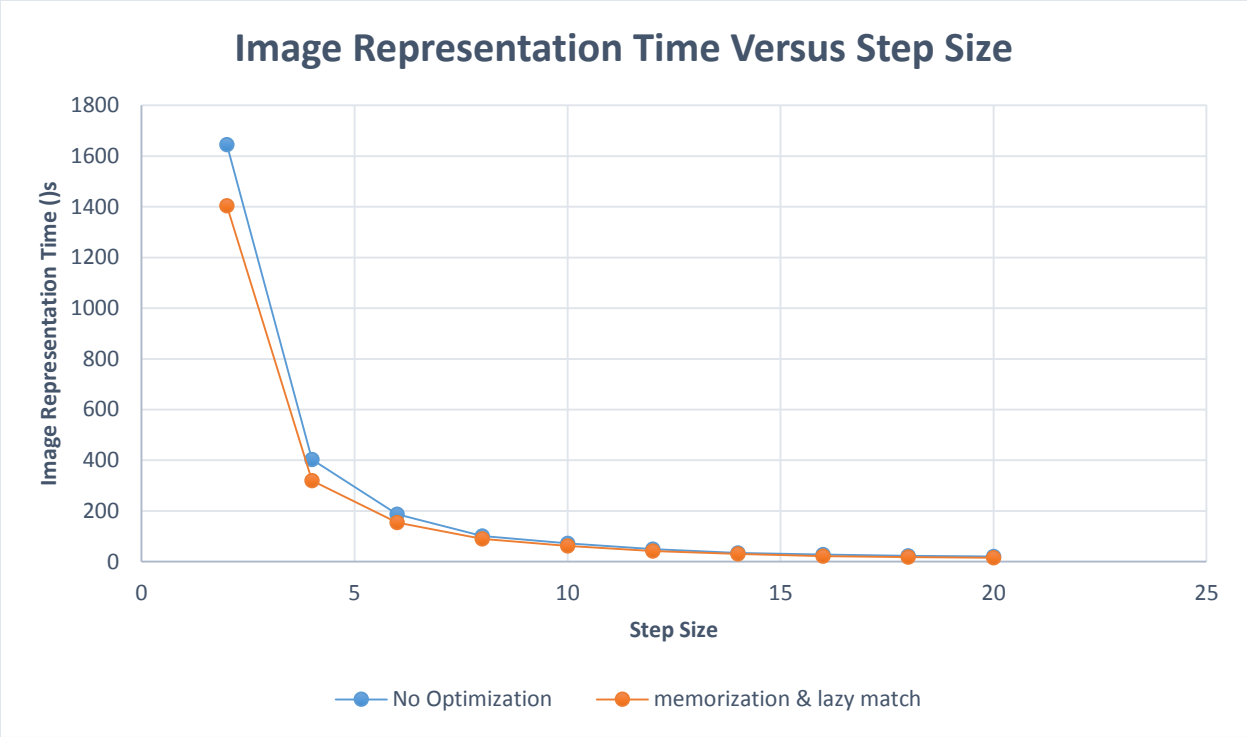
subset ratio	***edge***	***TP***	***FP***	***nP***	***nN***	***True Postive***	***False Positive***	***Recall***	**1 – *Precision***
0.1	canny	161	9	250	250	64.40%	3.60%	64.40%	5.29%
0.2	canny	166	12	250	250	66.40%	4.80%	66.40%	6.74%
0.3	canny	163	11	250	250	65.20%	4.40%	65.20%	6.32%
0.4	canny	172	10	250	250	68.80%	4.00%	68.80%	5.49%
0.5	canny	181	11	250	250	72.40%	4.40%	72.40%	5.73%
0.6	canny	179	10	250	250	71.60%	4.00%	71.60%	5.29%
0.7	canny	177	11	250	250	70.80%	4.40%	70.80%	5.85%
0.8	canny	172	9	250	250	68.80%	3.60%	68.80%	4.97%
0.9	canny	170	9	250	250	68.00%	3.60%	68.00%	5.03%
1	canny	169	10	250	250	67.60%	4.00%	67.60%	5.59%



Plot 3: Detect performance versus vocabulary search subset ratio



Plot 4: Algorithm run time versus vocabulary search subset ratio



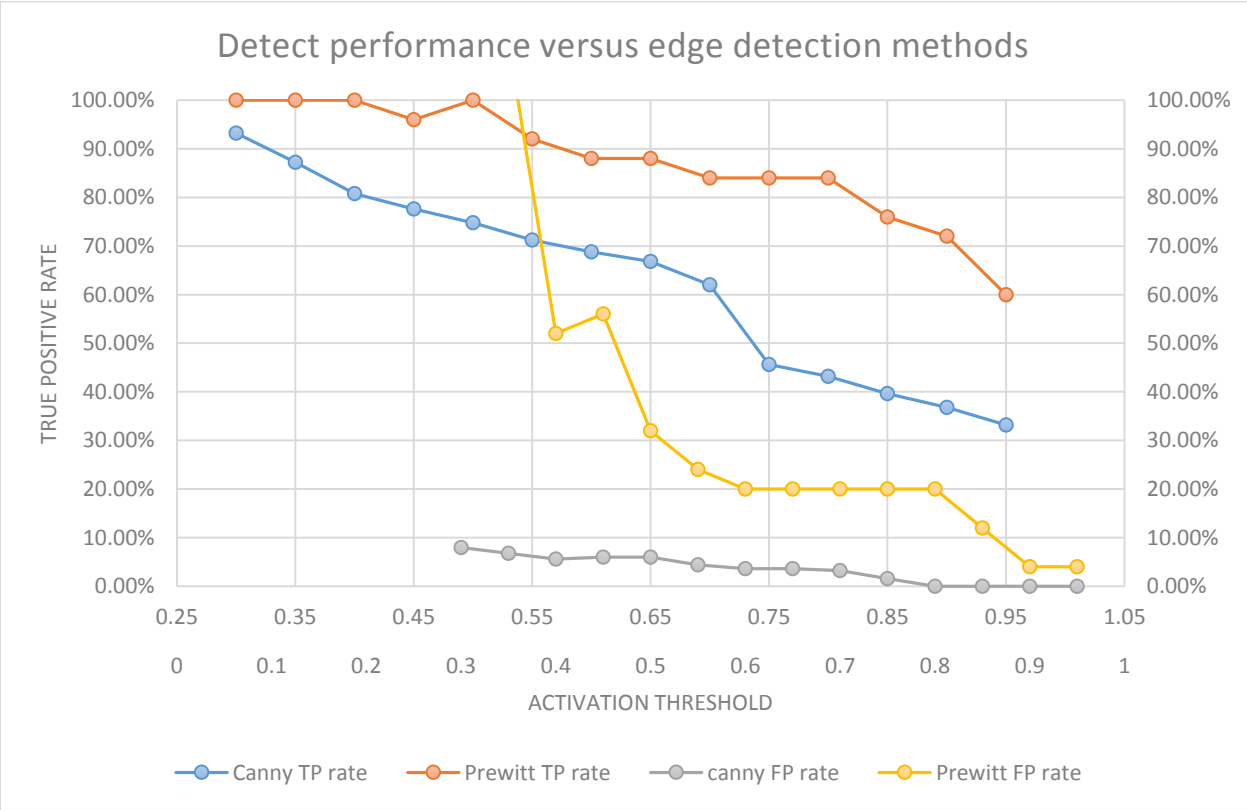
Plot 5 Image Representation Time versus Step Size

7.2 Edge Detector Results

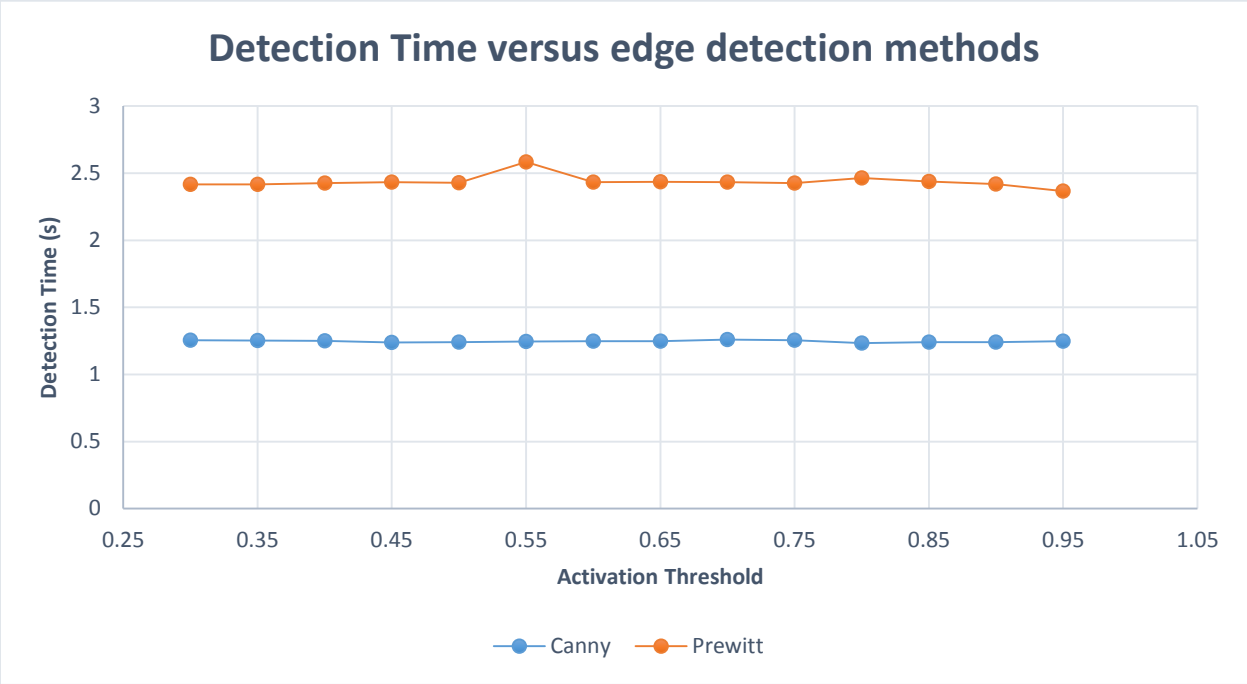
We tested two edge detection methods: canny edge detection and prewitt edge detection. In order to compare two performances, we introduce a variable called activation threshold. The object only deem to exist in the sub-image when the classifier's confidence level is above activation threshold, thus higher activation threshold, less false positive classification would be made. As we can see from Plot 6, for canny detector TP rate decrease below 10%.

On the other hand, for prewitt detector, the TP rate stays much higher than canny detector's TP rate and decrease more slowly with increasing of activation threshold.

However, the FP rate is extremely high, with above 100% when activation threshold lower than 0.45, and decreases almost exponentially down to 5% when activation threshold is 0.95. With larger activation threshold, prewitt detector perform much better in TP rate but suffer from higher FP rate, also the computation time required for prewitt is about twice large as canny detector as shown in Plot 7. As the result, we can use canny detector for fast processing speed and low FP rate, but use prewitt detector for much better TP rate.



Plot 6: Detect performance versus edge detection methods



Plot 7: Detection time versus edge detection methods

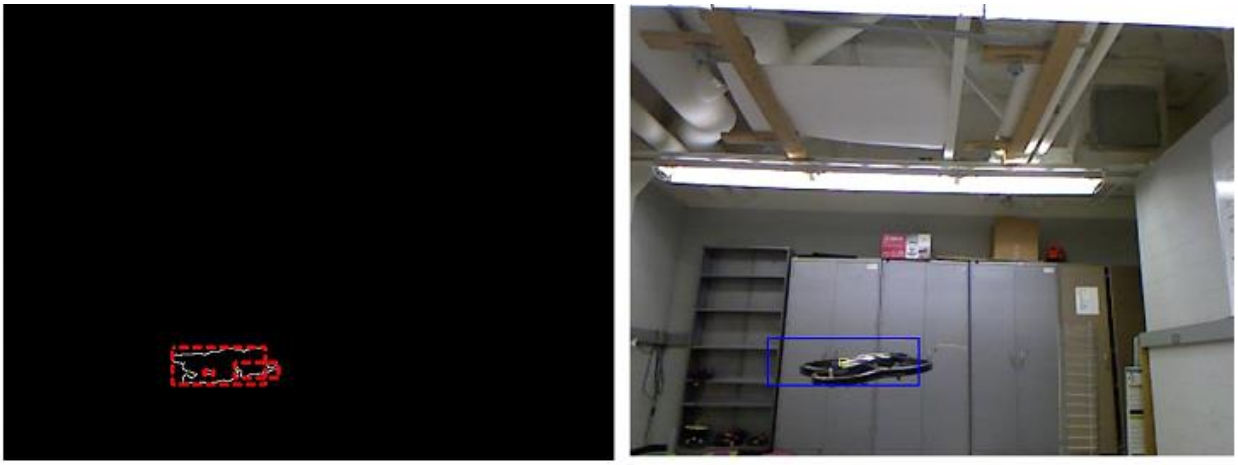
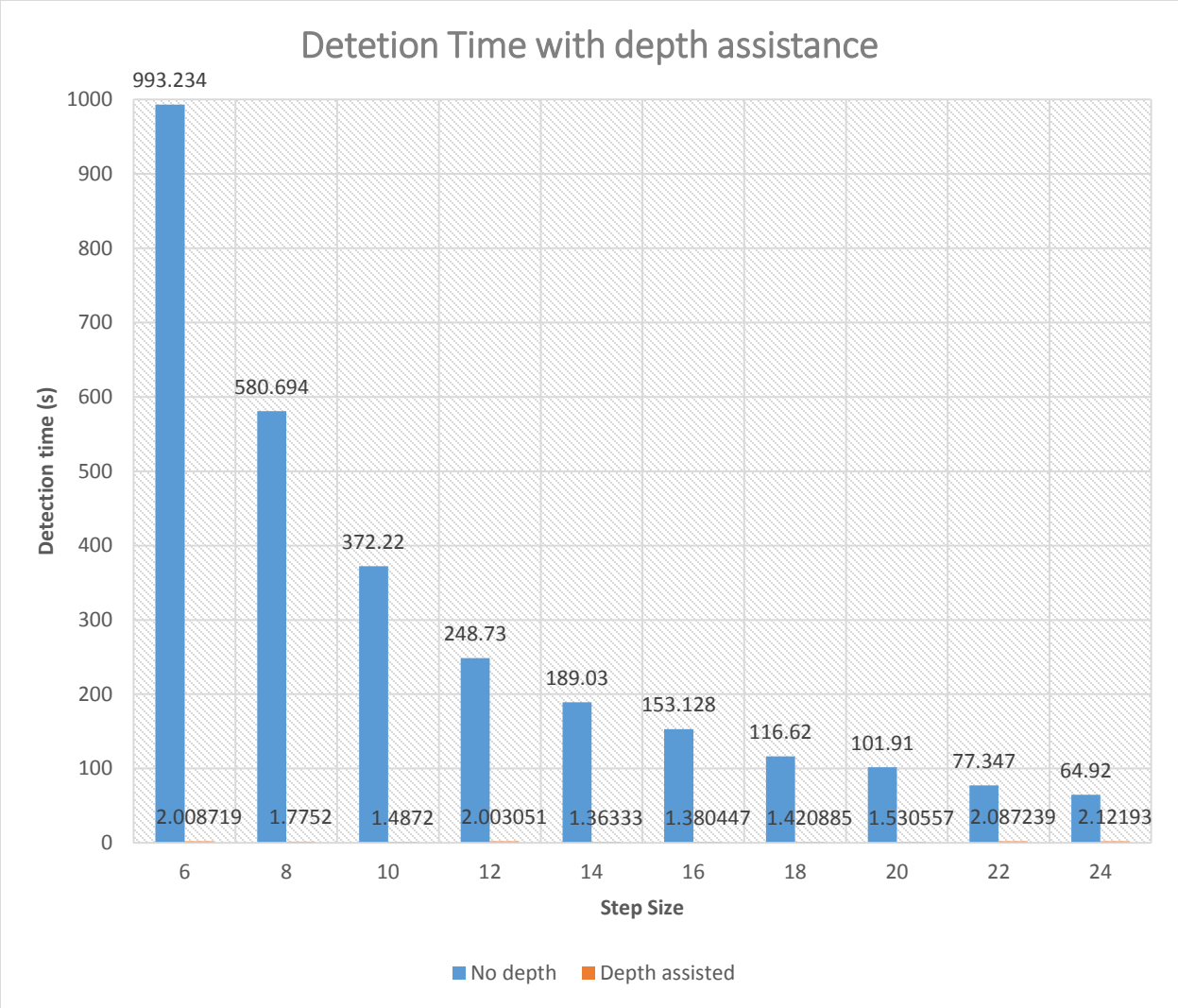


Figure 16: Vehicle detection with canny edge detector

Figure 17: PX4 Drone detected and located with aided of edge detector

7.3 Depth Aided Performance Comparison

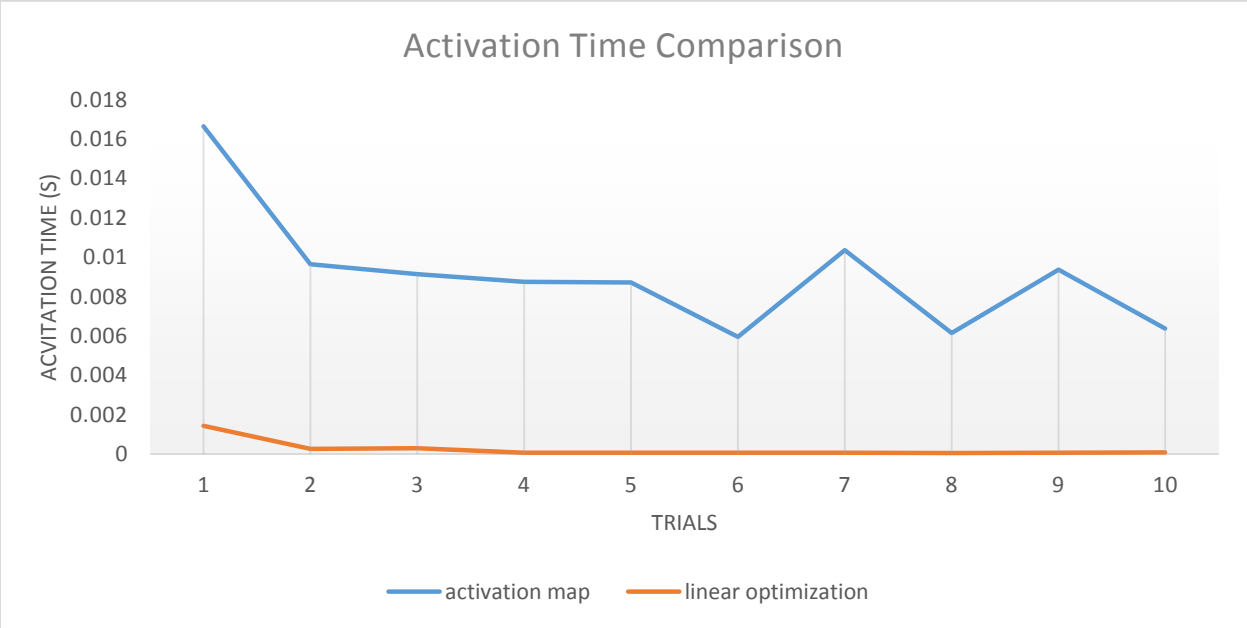
The depth-aided detection method gives us significant improvement in terms of speed while maintain high detection accuracy and low false positive detection rate. The detection time with depth assistance result can best be shown in the following Plot, the blue bar is the time required to detect an object in the given image, and the tiny hard-to-see yellow bar is the time required to detect an object with depth assistance.



Plot 8: Detection Time with depth assistance

7.4 Linear Neighbor Suppression

With Linear Neighbor Suppression Algorithm, the run time of neighbor suppression is also decreased. With 10 trials, the linear neighbor suppression algorithm consistently runs faster than the original algorithm by about 10 times. The result is presented in Plot.



Plot 9: Activation time comparison

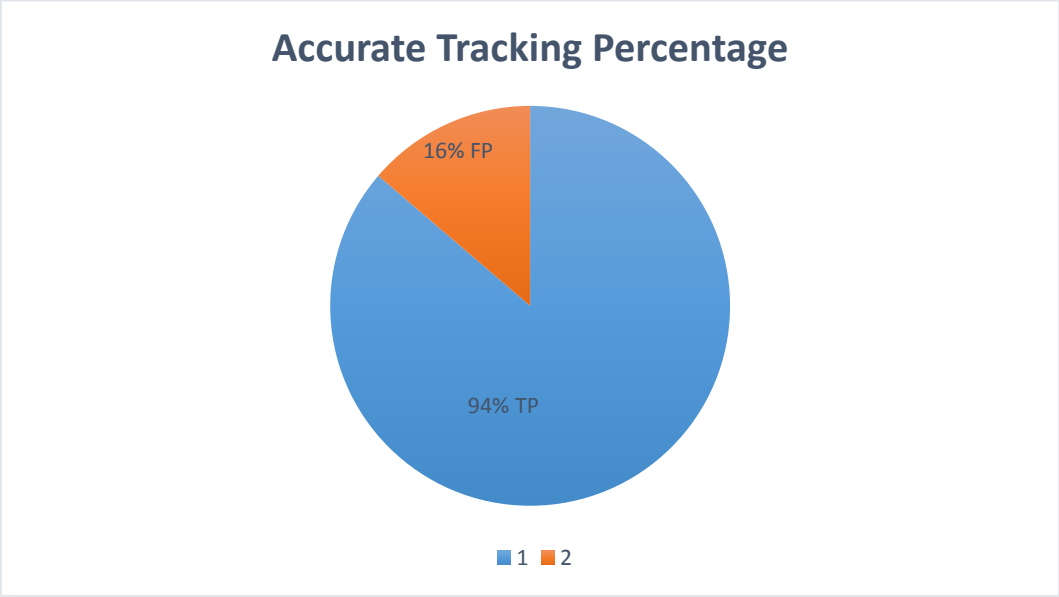
7.5 Overall Detection Accuracy and Performance

The Lazy Vocabulary Matching with Memorization gives marginal speed increase overall and in some cases much better performance as we expected. After comparing two edge detection methods and tuning several parameters, we found optimal combination of algorithms and parameters that gives us high detection accuracy with real-time performance. With help of Kinect sensor, we proved that

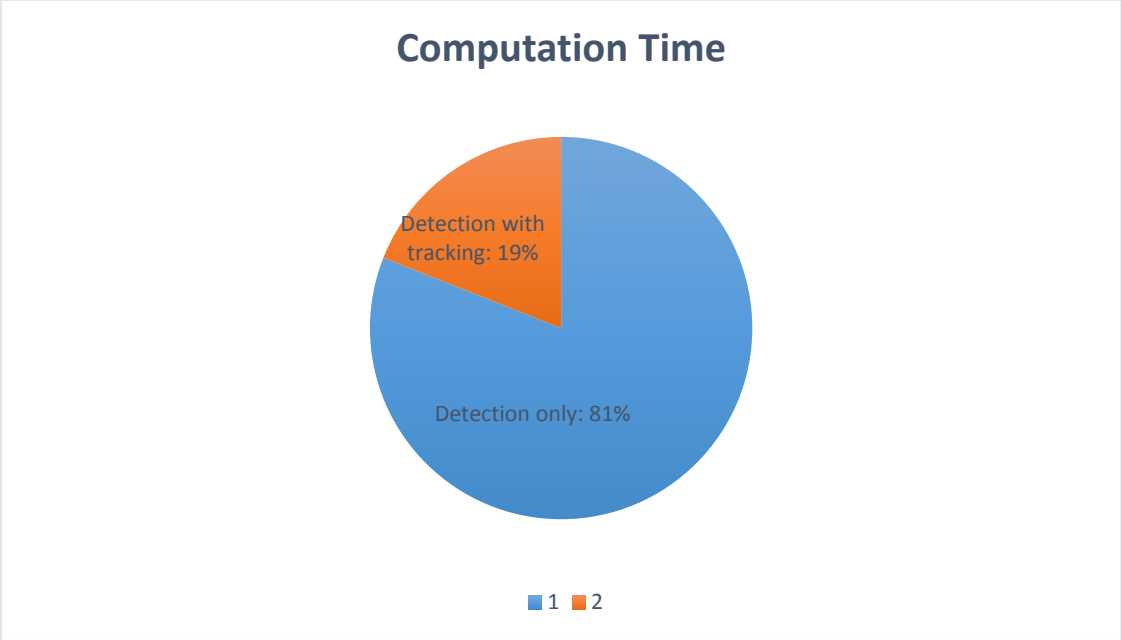
our speculation of real-time flying vehicle recognition and tracking is achievable with relatively low cost hardware and require no special feature points attachment.

7.6 Tracking Performance

The KLT algorithm tracking performance is measured by the tracking accuracy, which also uses true positive rate TP, which is number of vehicles' locations get accurately tracked, another is false positive rate FP, this is the number of vehicles' locations got inaccurate. The 94% TP as shown in the following pie chart means it gives higher localization accuracy than vehicle detection algorithm does, also it only takes about a quart of computation time than vehicle detection algorithm, which means using KLT tracker not only increase vehicle localization accuracy, it also makes overall computation more efficient.



Plot 10: Tracking performance measurement pie chart



Plot 11: Computation time comparison with and without tracking

Chapter 8: Conclusion

To summarize, we have presented a method to locate and track indoor flying object in real-time. We use sparse part-based image representation approach to recognize object and use the power of Kinect 3-D sensor to significantly increase detection speed. We also revised several algorithms including Lazy Vocabulary Matching with Memorization, which uses lazy search and memorization technology in computer algorithm to marginally increase performance. We use depth data from Kinect to conduct image preprocessing by edge detection method to per-locate the possible locations of vehicles. By only searching at several possible locations instead of iterating over entire image, it tremendously increase algorithm speed and still maintain excellent detection performance. Significant further improvements were also achieved by revising the original Neighbor Suppression Algorithm to avoid complex and time-consuming matrix operation. With data preprocessing, number of activation is small, and with linear neighbor suppression method we presented, we not only shrink down memory usage from 2-D to 1-D, we also increase algorithm speed by 10 fold. We presented a critical evaluation of our approach. For each algorithm we revised and presented, we conducted rigorous analysis on both the detection accuracy performance and corresponding algorithm run time. We showed that without sacrifice detection performance, we can tremendously decrease algorithm runtime and make detection and tracking in real-time. The future works includes further increase detection speed, add multi-scale and multi-class support, and possibly use the new Kinect 2 as our depth sensor, which is more advanced and more powerful.

References

Agarwal, S., & Roth, D. (2002). Learning a sparse representation for object detection. *7th European Conference on COmputer Vision-Part IV, ECCV*, 18.

Agarwal, S., Awan, A., & Roth, D. (2004, November). Learning to detect objects in images via a sparse, part-based representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26.

Bergen, J., Anandan, P., Hanna, K., & Hingorani, R. (1992). Hierarchical model-based motion estimation. *European Conference on Computer Vision*, 237-252.

Blum, A. (1992). Learning boolean functions in an infinite attribtue space. *Machine Learning*, 373-386.

Hwangbo, M., Kim, J.-S., & Kanade, T. (2009). Inertial-aided klt feature tracking for a moving camera. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 8.

Intel integrated performance primitives software, version 8.0, 8.0. (2014, December 15). (Intel) Retrieved December 15, 2014, from Intel Developer: https://software.intel.com/en-us/intel-ipp

Littlestone, N. (1991). Redundant noisy attributes, attribute errors, and linear threshold learning using Winnow. *4th Annu. Workshop in Comput. Learning Theory*, 147-156.

Weber, M., Welling, M., Perona, P. (2000). Unsupervised learning of models for recognition. *Sixth European Conference on Computer Vision*, 18-32.

Stubbs, A., & Dullerud, G. E. (2001). Networked Control of Distributed Systems: A testbed. *ASME International Mechanical Engineering Congress & Exposition*.

Grimson, W.E.L., Lozano-Perez, T (1985). Recognition and localization of overlapping parts from sparse data in two and three dimensions. *IEEE International Conference on Robotics and Automation*, 61-66.

Yang, M.-H., Roth, D., & Ahuja, N. (2000). Learning to recognize 3d objects with snow. *6th European Conference on Computer Vision-Part I*, 16.

Appendix A: Software Installation Guide

- **Hardware requirement**:
  Intel Pentium 2 core, 2GB ram and 64GB hard disk storage. You can get a similar machine at UIUC Surplus.

- **Operating System**:
  Ubuntu 14.04 64-bit desktop version, free for download at http://www.ubuntu.com/download/. The default username is: hotdrone, password: drone123. It has root privilege enabled and you can enter root by *sudo* command

- **Build-Essential**:
  After install Ubuntu 14.04, the first thing to do is to install latest development package from Ubuntu by typing following commands:
  - *Sudo apt-get update*
  - *Sudo apt-get upgrade*
  - *Sudo apt-get install build-essential*

- **Intel IPP 7.1**:
  The Intel Integrated Performance Primitives should be install right after installing essential building package from Ubuntu. It has required shared library needed for OpenCV. The software is available from Kinect Vision Server Library folder, but also available for download from Intel development website. (The product serial number: **NG7L-Z9VJB95V**)
  - *sudo apt-get install gcc-multilib*
  - *Run ./install.sh at IPP7.1 folder*
  - *Edit ~/.bashrc*
    - *# My .bashrc*
    - *Export IPPROOT=/opt/intel/composer_xe_2013.1.117/ipp*
    - *Export LIBRARY_PATH=$LIBRARY_PATH:/* opt/intel/composer_xe_2013.1.117/ipp/lib/intel64:/opt/intel/composer_xe_2013.1.117/compiler/lib/intel64
    - Export *LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/composer_xe_2013.1.117/ipp/lib/intel64:/opt/intel/composer_xe_2013.1.117/compiler/lib/intel64*

- **Libfreenect**:
  This library is required to run Kinect and use Kinect C++ library for image retrival. It can be downloaded for free from https://github.com/OpenKinect/libfreenect,
  - *Sudo apt-get install git-core cmake libglut3-dev pck-config build-essential libxmu-dev libxi-dev libusb-1.0-0-dev*
  - *Git clone git://github.com/OpenKinect/libfreenect.git*
  - *Cd libfreenect*
  - *Mkdir build*

- *Cd build*
- *Cmake ..*
- *Make*
- *Sudo make install*
- *Sudo ldconfig /usr/local/lib64*
- *Sudo glview*
- *Sudo adduser $USER video*
- *Sudo nano /etc/udev/rules.d/51-kinect.rules*
- *Edit the file as described in http://openkinect.org/wiki/Getting_Started*

- **OpenCV 2.4.10**:
  The OpenCV 2.4.10 is the library version used for aerial vehicle detection and tracking software. Any version higher or lower might results in software incompatibility.
  - *Download the OpenCV source code either from Github or from local Kinect Vision Server.*
  - *Sudo apt-get install libtiff4-dev, libjasperm ligtbb-dev libdc1394-22-dev libavcodec-dev libavformat-dev libswscale-dev libtheora-dev libvorbis-dev libxvidcore-dev libx264-dev vasm libfaac-dev libopencore-amrnb-dev libopencore-amrwb-dev libv4l-dev libxine-dev python-dev python-numpy python-tk*
  - *Mkdir build*
  - *Cd build*
  - *Cmake-gui ..*
  - *Click configure, enables OpenNI, PNG, IPP, FAST MATH*
  - *Make* –j
  - *Sudo make install*

- **Zmq 2.X**:
  All Zmq 2.x legacy versions correctly support aerial vehicle detection and tracking algorithm communication system. It can be downloaded from zeromq.org.
  - *Download the ZMQ package from official website or from local Kinect Vision Server*
  - *./Configure*
  - *Make*
  - *Sudo make install*

- **PX4 QGroundControl Station**:
  PX4 QGroundControl v2.0 is the default version used for PX4 aerial vehicle communication central and the tool for PX4 quadcopter sensor tuning and control parameters tuning.
  - *Sudo apt-get-repository ppa:beineri/opt-qt54-trusty*
  - *Sudo add-apt-repository ppa:qgroundcontrol/ppa*
  - *Sudo apt-get update && sudo apt-get install qgroundcontrol*

Appendix B: SNoW Classifier User Guide

- **Mode Selection**:
  - *–train: The system is run in training mode and input file is considered to be a set of labeled training examples*
  - *–test: The system is run in a batch test mode. The input file can be labeled or unlabeled testing examples. Each example in the file is presented to the system and classified.*
  - *–evaluate: The system is run in an interactive test mode. A single labeled or unlabeled example is supplied on the command line. The process terminates after making a prediction for this single example.*
  - *–server: The system is run in server mode. It continuously accepts input samples from network connections and reply with a prediction or messages.*

- **Architecture Definition**:
  - *–A <architecture file>: specifies the name of a file from which to read the desired architecture definition and parameters*
  - *–B :targets : specifies targets that will be trained using the naïve Bayes algorithm*
  - *–P <learning rate>, <threshold>, >default_weight>:targets :specifies targets that will be trained using the single layer perception algorithm.*
  - *–W <promotion>, <demotion>, <threshold>, <default_weight>:targets : specifies targets that will be trained using the winnow algorithm along with their parameters.*

- **I/O Options**:
  - *–a <+ | ->: specifies whether to use highly accurate weights for features or to approximate them when writing out the network.*
  - *–c <i>: In training mode. The interval, in the number of examples presented, at which to output a snapshot of the network.*
  - *–E <errorfile>: specifies the name of a file in which to write information about mistakes during testing.*
  - *–F <networkfile>: specifies the name of a file in which the resulting networks are written to or read from.*
  - *–I <inputfile>: specifies the input file from which examples are read.*
  - *–O <accuracy | winners | allpredictions | allactivations | allboth>: specifies which output mode to use when reporting results during test mode.*
  - *–R <results_file>: specifies the name of a file in which the results of testing are output.*
  - *–T <testing_file>: using this option a network can be trained and tested in the same invocation of snow.*

Appendix C: PX4 Quadcopter Guide

- **PX4 Toolchain Installation**:
  You need to install PX4 toolchain in order to compile your code and flash it to the px4 flight management control board. The PX4 toolchain can be installed on Linux or Windows. Here is the installation guide to install on Ubuntu 14.04
  - Sudo apt-get update
  - Sudo apt-get install python-serial python-argparse opencd flex bison libncurses5-dev autoconf texinfo build-essential libftdi-dev libtool zlib1g-dev git-core wget zip python-numpy
  - sudo apt-get install libc6:i386 libgcc1:i386 gcc-4.6-base:i386 libstdc++5:i386 libstdc++6:i386
  - sudo usermod –a –G dialout $USER
  - pushd .
  - cd ~
  - wget https://launchpadlibrarian.net/186124160/gcc-arm-none-eabi-4_8-2014q3-20140805-linux.tar.bz2
  - tar –jxf gcc-arm-none-eabi-4_8-2014q3-20140805-linux.tar.bz2
  - exportline=”export PATH=$HOME/gcc-arm-none-eabi-4_8-2014q3/bin:\$PATH”
  - if grep –Fxq “$exportline” ~/.profile; then echo nothing to do; else echo $exportline >> ~/.profile; fi
  - . ~/.profile
  - Popd

- **Building and Flashing Software**:
  The software is located at px4-firmware-AR folder, cd into that folder and execute following command to compile and flash the code to the PX4 flight management control board. You can get detailed PX4 building guide at PX4 developer home page.
  - Make distclean
  - Make archives
  - Make –j8
  - Make –j8 px4fmu-v1_default
  - Make upload px4fmu-v1_default